Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Živković

Generiranje ogrodja za izdelavo vmesnikov za namensko programiranje

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Luka Šajn

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Izdelajte aplikacijo, ki bo poenostavila in pohitrila izdelovanje REST API aplikacij tako, da generira ogrodje projekta z vso pripadajočo izvorno kodo, dokumentacijo in testi. Generiranemu projektu je potrebno dodati le še implementacijo samih API klicev, s čimer projekt postane funkcionalna spletna REST API aplikacija. Uporabljeno naj bi bilo čim več obstoječih orodij, aplikacija sama pa naj bo nadvse enostavna za uporabo.

Zahvaljujem se Mihi Vidmarju za idejo, Mateju Spiller Muysu za ideje in nasvete pri izdelavi aplikacije in reševanju problemov, Alešu Volčini za formuliranje diplome, mentorju za pomoč pri pisanju ter družini za spodbudo.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
POC	proof of concept	dokaz koncepta
API	application programming interface	vmesnik za namensko programiranje
REST	representational state transfer	prenos reprezentančnega stanja
IDE	integrated development environment	integrirano razvojno okolje
HTTP	hypertext transfer protocol	protokol za prenos informacij po spletu
RBAC	role based access control	nadzor dostopa s pomočjo vlog
SQL	structured query language	strukturirani povpraševalni jezik
CRUD	create, read, update, delete	kreiraj, beri, posodobi, izbriši

Povzetek

Naslov: Generiranje ogrodja za izdelavo vmesnikov za namensko programiranje

Avtor: Jan Živković

Ker se v industriji dela veliko POC projektov in se za začetno razvijanje porabi veliko časa, aplikacije pa niso vedno uspešne, sem izdelal program, ki z uporabo konfiguracijske datoteke generira aplikacijo s potrebnim API vmesnikom. Aplikacija vsebuje podatkovno bazo, do katere lahko dostopamo preko spletnih API klicev. API klicem lahko po želji omejimo dostop glede na uporabnika, pogostost in število hkratnih dostopov do API klicev. Lastnik storitve lahko spremlja statistiko uporabe API klicev. Razvijalec mora generiranemu projektu dodati še samo implementacijo. Rezultat diplomskega dela je uspešna generacija API aplikacije z uporabo konfiguracijske datoteke, ki generira celoten projekt, torej izvorno kodo, dokumentacijo in teste.

Ključne besede: api, generator, spletna aplikacija, dokumentacija, testi.

Abstract

Title: Generation of API Framework

Author: Jan Živković

Because we spend alot of time developing POC projects and those projects aren't always successful, I have created a program, that will be able to generate API application using a simple configuration file. This application contains a database that can be accessed through web API calls. If required, API calls can be restricted by user, frequency and number of concurrent access to API calls. The service owner can monitor the statistics of API calls to insure its best posible operation. The Developer needs to add the implementation to the generated project. Result of this thesis is successful generation of API application using the configuration file, which creates the whole project, this includes the source code, documentation and tests.

Keywords: api, generator, web application, documentation, tests.

Poglavje 1

Uvod

Ker se v svetu čedalje več uporabljajo spletne aplikacije, kot so REST servisi, je za hitro prototipiranje aplikacij potrebno orodje, ki le-to omogoča. V diplomskem delu sem izdelal orodje, ki omogoča hitrejšo izdelavo REST servisov in s tem pohitril ter poenostavil izdelavo novih spletnih produktov. Orodje generira ogrodje aplikacije z vso pripadajočo izvorno kodo, dokumentacijo in s testi, ki jih lahko razvijalci enostavno implementirajo in testirajo. Za izdelavo aplikacije sem uporabil precejšen nabor že obstoječih knjižnic in izdelal primer API aplikacije z uporabo najboljše prakse pri razvijanju le-teh.

Poglavje 2

Raziskovanje razpoložljivih orodij za izdelavo API

V tem poglavju predstavljam, katere že obstoječe tehnologije lahko uporabim v izdelavi aplikacije in na kratko predstavljam izbrana orodja.

2.1 Orodja za izdelavo API

Ker sem za izdelavo aplikacije želel uporabiti in združiti čim več že obstoječih rešitev in s tem poenostaviti izdelavo aplikacije ter omogočiti enostavnejšo uporabo vsem razvijalcem, saj bi večina že znala uporabljati uporabljene knjižnice, sem začel iskati primerne knjižnice za uporabo v REST API. Izmed vseh rešitev, ki sem jih našel, so v končni izbor prišla naslednja orodja:

- `Swagger` [24],
- `Apiary` [4],
- `Mashape` [15],
- in `Api studio` [3].

Na koncu sem izbral Swagger, saj podpira generiranje izvorne kode, dokumentacije in testov v enem paketu. Za integracijo Swagger v Spring aplikacijskem ogrodju sem uporabil SpringFox [23].

2.1.1 Swagger

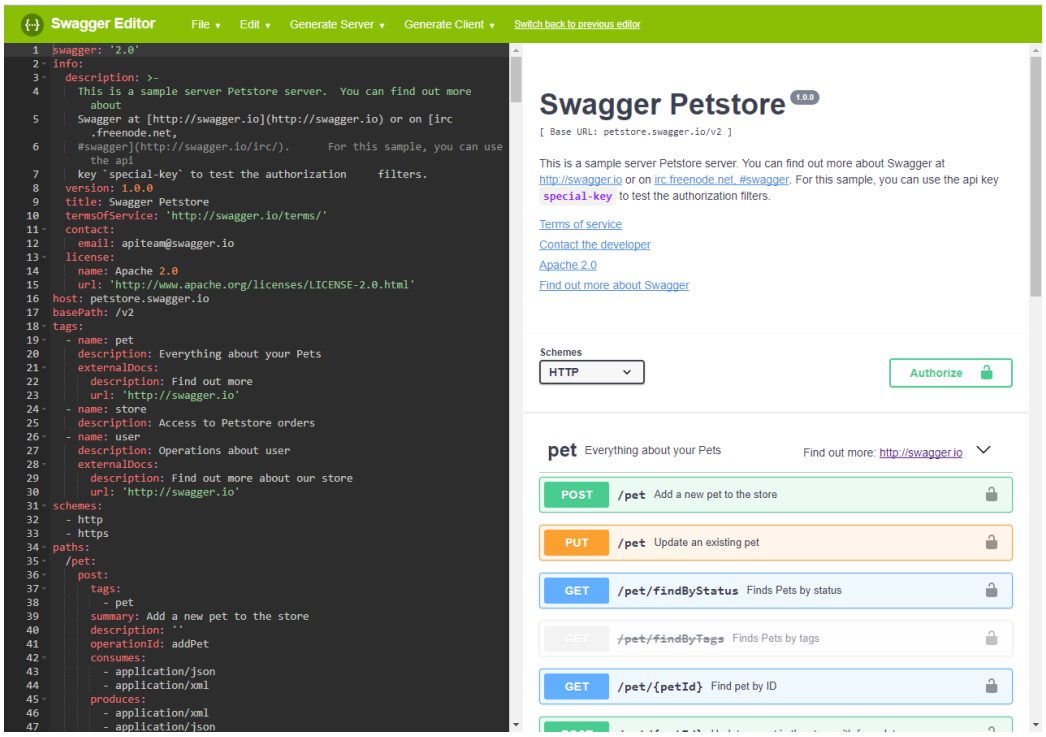
Swagger je set orodij za izdelavo API-jev, ki omogoča oblikovanje, dokumentiranje, testiranje in generiranje ogrodja aplikacije v velikem številu programskih jezikov. Uporablja OpenAPI specifikacijo, ki je standard za izdelavo API-jev in se uporablja v skoraj vseh programskih jezikih. Ta specifikacija omogoča, da lahko razvijalec razume definicijo API-ja brez težav. Prednost Swaggerja je, da je na voljo brezplačno in tudi, da je odprtokoden projekt, kar omogoča razvijalcem tako dodajanje novih funkcionalnosti kot tudi popravljanje obstoječih. S tem zagotavlja tudi nadaljnjim razvijalcem dobro podlago za nadaljevanje razvoja.

Swagger Editor

Swagger Editor je spletna aplikacija, ki omogoča lažjo izdelavo konfiguracijske datoteke, ki jo nato Swagger Codegen uporabi za izdelavo ogrodja aplikacije. Podpira dva formata za izdelavo konfiguracijskih datotek, ki sta:

- `YAML` [27]
- in `JSON` [13].

V Swagger Editor je potreben le enostaven klik z miško, da si generiramo strežniško ali klientsko izvorno kodo za vneseno konfiguracijsko datoteko, s čimer lahko takoj začnemo z razvijanjem API kot tudi s testiranjem le-tega.



Slika 2.1: Izgled Swagger Editorja

Swagger Codegen

Swagger Codegen je orodje, ki generira strežniško ali klientsko izvorno kodo za različne konfiguracijske datoteke. Izvorno kodo lahko generira za različne programske jezike, med njimi so tudi:

- Java,
- Go,
- Erlang,
- PHP,
- Javascript,
- Ruby,
- Perl,
- Python,

- Haskell,
- in ASP.NET.

2.1.2 Predloge

Za generiranje projekta, izvorne kode, dokumentacije in testov uporablja predloge, ki so napisane v knjižnici Mustache [17]. Mustache je sistem za generiranje preprostih spletnih predlog. Velja za "sistem brez logike", ker ne podpira pogojnih stavkov, zank in manipulacije podatkov. Zaradi tega je te predloge zelo težko uporabljati za generiranje kode, saj je treba vso logiko narediti v sami programski kodi, kjer pa to povzroča precejšne probleme, med drugim tudi kapitalizacije imen spremenljivk in razredov. V izseku kode 2.1 je podan primer predloge v Mustache, podatki za predlogo v izseku kode 2.2 in generirana koda v izseku kode 2.3.

Izsek kode 2.1: Primer Mustache predloge

```
{{person}}
  Ime: {{name}}
  Priimek: {{surname}}
  Starost: {{age}}
  {{#hobbies}}
    {{#-first}}Hobiji:{{/-first}}
      - {{.}}
  {{/hobbies}}
{{/person}}
```

Izsek kode 2.2: Primer vhodnih podatkov za predlogo

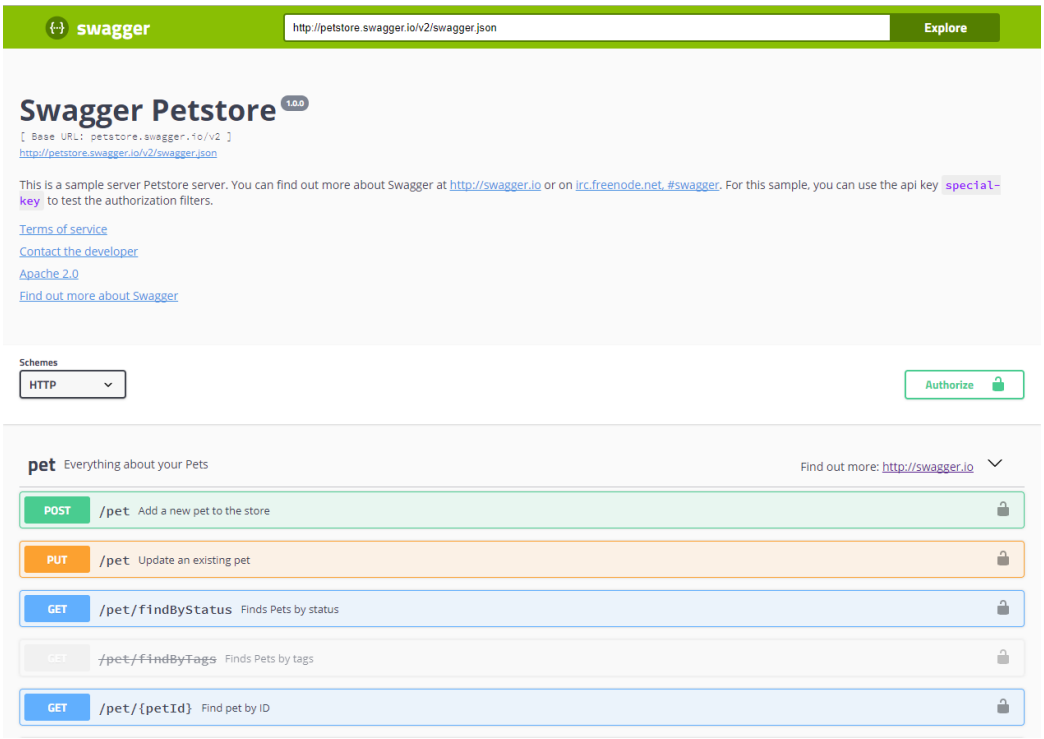
```
{
  "person": {
    "name: "Jan",
    "surname": "Zivkovic",
    "age": 22,
    "hobbies": {
      "programiranje", "igranje iger", "gledanje
        ↪ filmov"
    }
  }
}
```

Izsek kode 2.3: Primer generirane predloge

```
  Ime: Jan
  Priimek: Zivkovic
  Starost: 22
  Hobiji:
  - programiranje
  - igranje iger
  - gledanje filmov
```

Swagger UI

Swagger UI je spletna aplikacija, ki sestavi in prikaže dokumentacijo celotnega API. API klici so združeni v skupine tako, kot so definirani v izvorni kodi. S klikom na skupino odpremo prikaz pripadajočih API klicev. Za vsak API klic prikaže, kako se uporablja, možne statuse klicev API, potrebne modele, format sprejemanja in pošiljanja podatkov, način klica API itd.



Slika 2.2: Izgled Swagger UI

Poglavje 3

Orodja

Za izdelavo tega API ogrodja sem uporabil razna orodja, ki so mi olajšala izdelavo, prototipiranje in testiranje.

Uporabil sem:

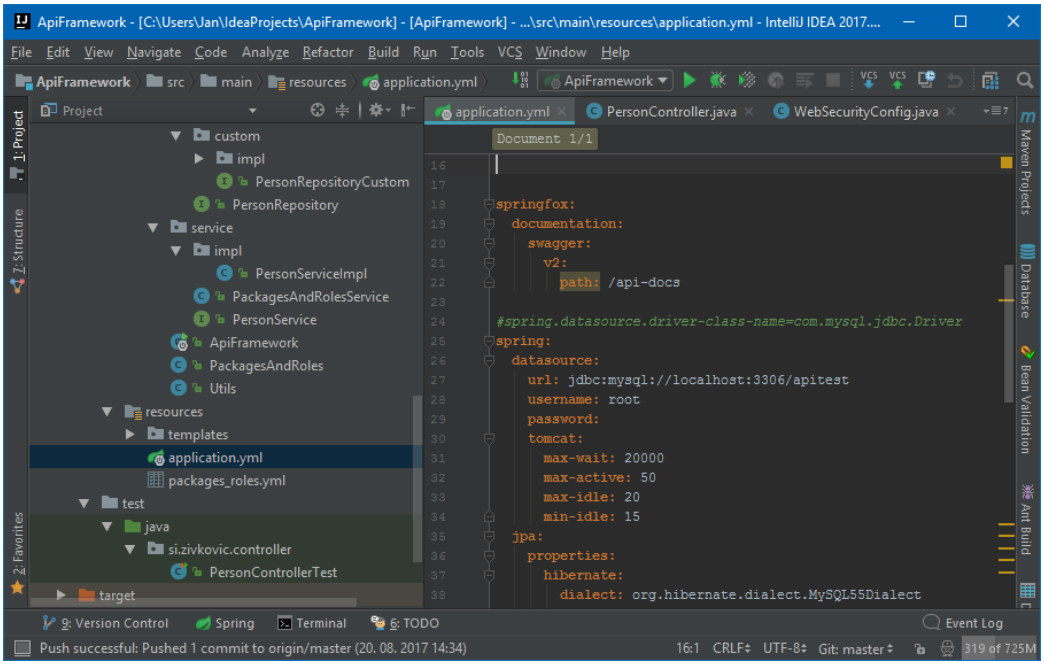
- `IntelliJ Idea` [10] - za programiranje,
- `Postman` [19] - za testiranje API klicev,
- `Notepad++` [18] - za vse (programiranje, zapiske ...),
- `Xampp` [26] - za MySQL bazo,
- `Git` [5] - za upravljanje izvorne kode,
- `Git-Bash` [6] - za poganjanje maven procesov v konzoli,
- `Maven` [16] - za upravljanje postopka gradnje aplikacije.

3.1 IntelliJ IDEA

IntelliJ IDEA je IDE, ki se uporablja za razvijanje v programskem jeziku Java. Sam uporabljam IntelliJ IDEA Ultimate, ki je plačljiv, vendar za študente brezplačen. Vsebuje tudi uporabne funkcionalnosti za razvijalce, kot so:

- predlogi dokončanja kode,
- pomoč z uporabljenimi aplikacijskimi ogrodji,
- integracija s sistemi za upravljanje z izvorno kodo,
- predlogi boljše strukture izvorne kode,
- preprečevanje slabe prakse razvijanja, kot je podvajanje kode,
- ...

Na sliki 3.1 je prikazan videz programa v temni izvedbi kot tudi sintaksno barvanje kode v yaml datoteki.



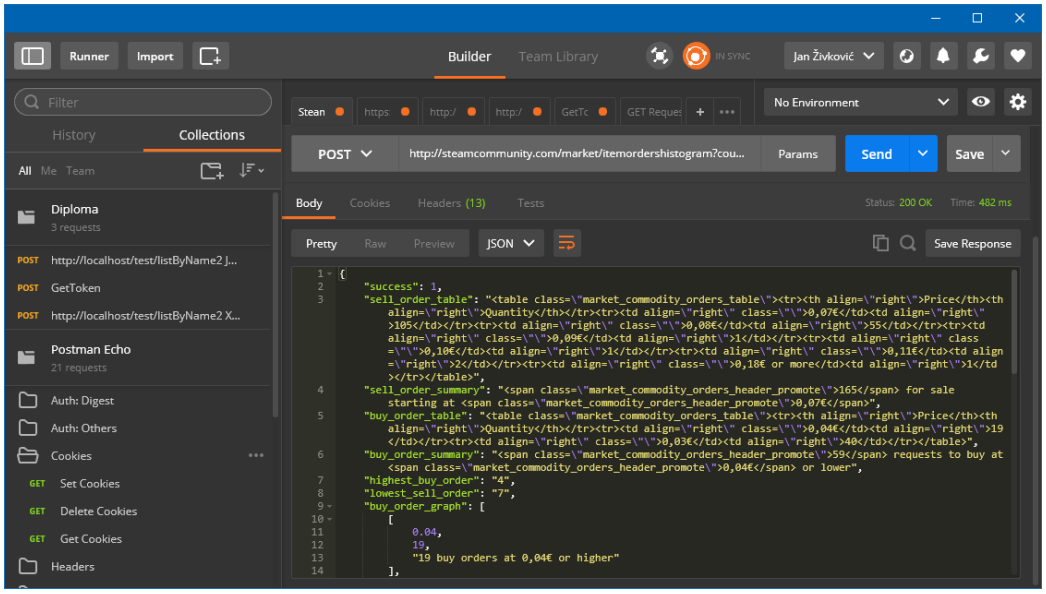
Slika 3.1: Izgled IntelliJ IDEA Ultimate

3.2 Postman

Postman je orodje, ki se uporablja za deljenje, testiranje, dokumentiranje in monitoriranje API-jev. Izvajanje API klicev s Postman aplikacijo je zelo enostavno, saj je uporabniški vmesnik intuitiven za uporabo. Med drugim podpira tudi formatiranje odgovora API, dodajanje HTTP glav in različne tipe avtentikacije, kot so:

- Basic auth,
- Digest auth,
- OAuth 1,
- OAuth 2,
- Hawk Authentication,
- AWS Signature,

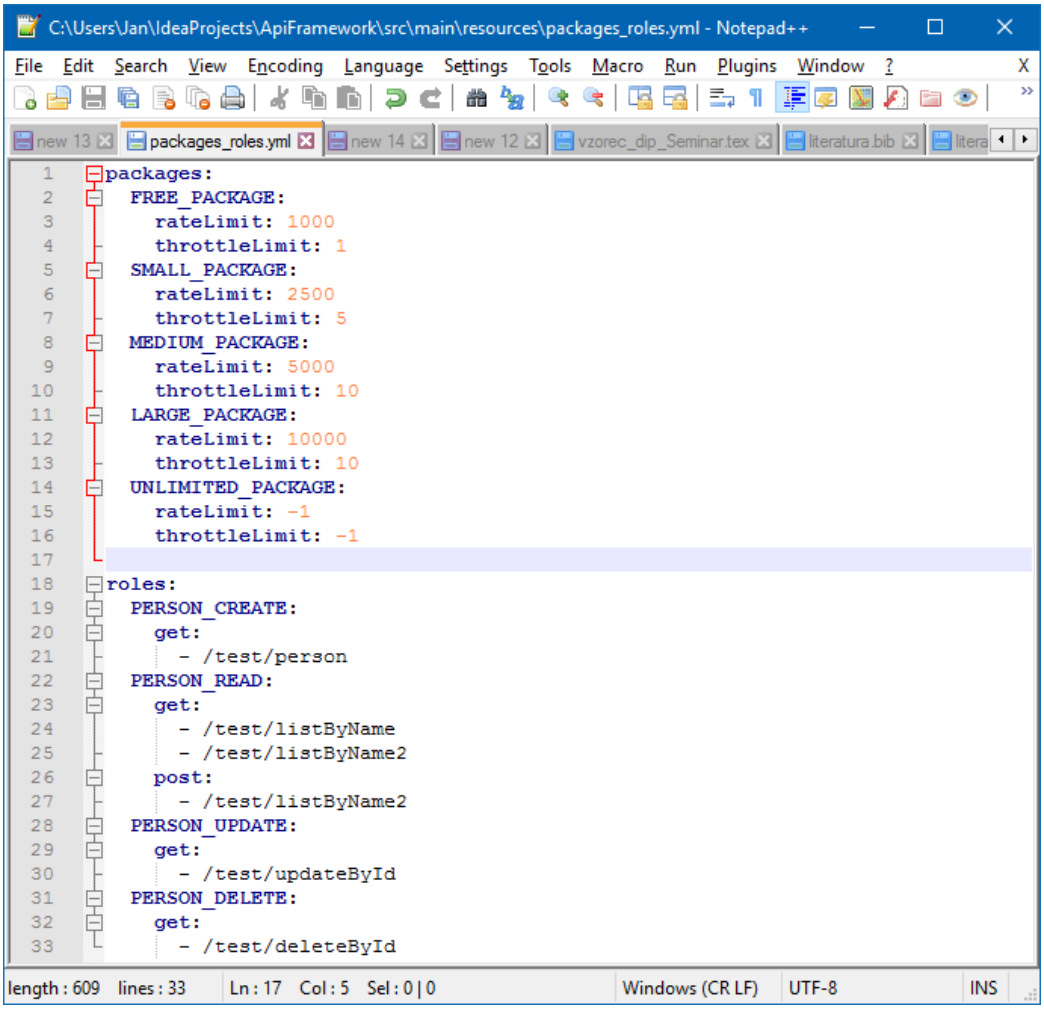
kot tudi izvažanje in uvažanje shranjenih API klicev. Na sliki 3.2 je prikazan videz programa in tudi primer odgovora API klica.



Slika 3.2: Videz Postman programa

3.3 Notepad++

Notepad++ je kot navadna beležnica, le da je ta na steroidih, lahko bi rekli, da je to beležnica za razvijalce, saj deluje zelo hitro in podpira skoraj vse programske jezike, formatiranje, barvno označevanje sintakse, zavihke, iskanje po mapah in datotekah in še veliko več s pomočjo vtičnikov. Notepad++ sem uporabljal za urejanje razne izvorne kode, od Java kode, Mustache predlog, xml datotek, yaml datotek in drugih uporabljenih formatov datotek. Na sliki 3.3 je prikazan program z odprto yaml datoteko.



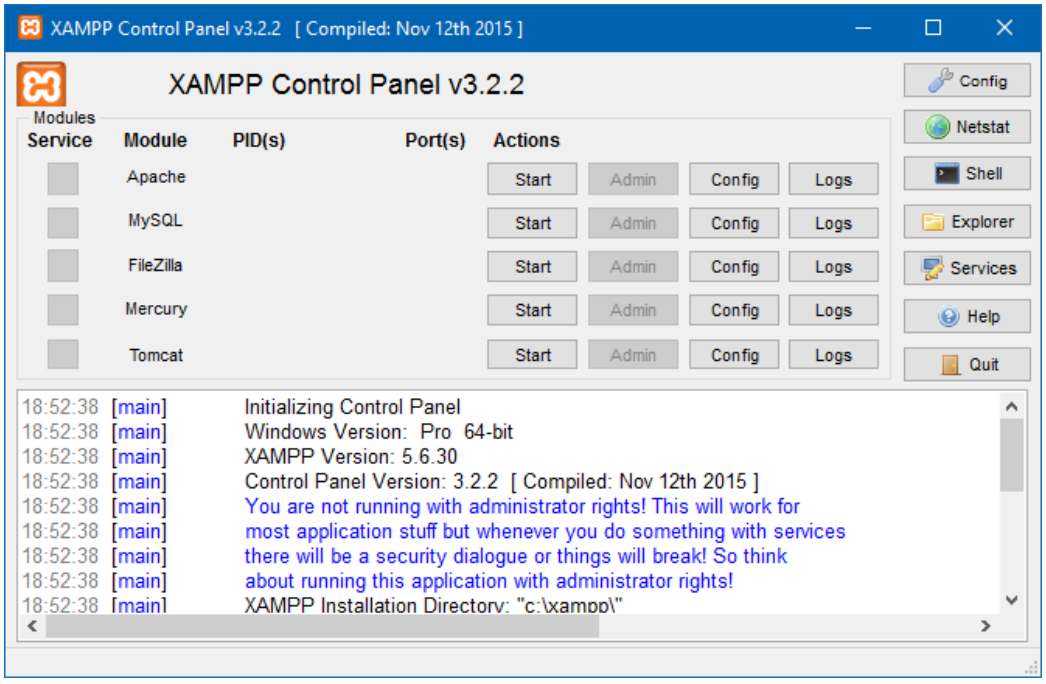
Slika 3.3: Izgled Notepad++

3.4 Xampp

Xampp je programski paket, ki vsebuje različna orodja, ki se uporabljajo za izdelavo spletnih strani. Vsebovana orodja so:

- Apache HTTP strežnik,
- MySQL baza,
- FileZilla,
- Mercury,
- in Tomcat.

Za Xampp sem se odločil zato, ker olajša namestitev MySQL baze, vendar sem s tem pridobil še druga orodja, za katera nisem imel uporabe. Na sliki 3.4 je prikazana nadzorna plošča programa, preko katere se zaženejo ali ustavijo različna vsebovana orodja.



Slika 3.4: Xampp nadzorna plošča

3.5 Git

Git je sistem za upravljanje različic izvorne kode, ki omogoča, da lahko več ljudi dela na istem projektu. Omogoča avtomatsko združevanje kode, vejanje, primerjanje itd. Napisal ga je Linus Torvalds skupaj z ostalimi razvojniki za potrebe razvijanja jedra Linux operacijskega sistema. Git je zelo razširjen in uporabljan sistem za upravljanje izvorne kode. Sam sem uporabljal Git preko programa IntelliJ IDEA.

3.5.1 Git Bash

Git Bash je emulacija Linux ukazne lupine, preko katerega sem kreiral Git repozitorij in poganjal Maven komande. Uporabil sem ga zato, ker sem Linux ukazne lupine bolj navajen kot klasične Windows ukazne vrstice. Na sliki 3.5 je prikazan videz programa.

Slika 3.5: Videz Git Bash programa

3.6 Maven

Maven je sistem za vodenje in gradnjo projektov. Upravlja različne gradnike, kot so vtičniki, postopki za gradnjo projekta in vsebuje knjižnice, ki jih projekt potrebuje za delovanje. Vsi vtičniki, postopki, knjižnice in drugi elementi so definirani v datoteki pom.xml, katere primer je viden spodaj v izseku kode 3.1.

Izsek kode 3.1: Primer pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema
           ↪ -instance"
         xsi:schemaLocation="http://maven.apache.org
           ↪ /POM/4.0.0 http://maven.apache.org/xsd
           ↪ /maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>si.zivkovic</groupId>
    <artifactId>ApiFramework</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</
          ↪ artifactId>
        <version>1.5.3.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</
              ↪ groupId>
            <artifactId>spring-boot-starter-web</
              ↪ artifactId>
        </dependency>
    </dependencies>

</project>
```

Poglavje 4

Izdelava primera generirane API aplikacije

V tem poglavju predstavljam primer želenega API vmesnika, kot bi ga potrebovala neka aplikacija. Na tem primeru sloni izdelava generatorja, ki generira izvorno API kodo glede na podano konfiguracijsko datoteko.

4.1 Priprava projekta

Za izdelavo projekta sem uporabil programski jezik Java, saj so vsi generatorji definirani v tem programskem jeziku. V IntelliJ Idea sem kreiral nov Maven projekt in začel dodajati obstoječe knjižnice, ki jih potrebujem za izdelavo tega. Dodal sem Spring [21] aplikacijsko ogrodje, Hibernate [9], MySQL-Connector, Swagger [24], Apache Commons in Log4j2[1], ki je privzeto orodje za beleženje v Spring.

Spring je aplikacijsko ogrodje, ki sem ga uporabil za izdelavo tega projekta. Spring knjižnice, ki sem jih uporabil, in razlogi za uporabo so:

- `spring-boot-starter-parent` - vsebuje osnovne gradnike za aplikacijo,

- `spring-boot-starter-web` - vsebuje gradnike za izdelavo spletne aplikacije,
- `spring-boot-starter-data-jpa` - vsebuje gradnike za povezavo spring z hibernate,
- `spring-boot-starter-security` - vsebuje gradnike za varnost, kot so avtentikacija/avtorizacija in v mojem primeru basic auth,
- `spring-boot-starter-thymeleaf` - vsebuje pogon za izdelavo predlog Thymeleaf,
- `spring-boot-starter-test` - vsebuje gradnike za izdelavo testov,
- `spring-tx` - vsebuje gradnike za transakcije,
- `spring-orm` - vsebuje gradnike za objektno-relacijsko preslikovanje,
- `Hibernate` - je objektno-relacijska knjižnica
- `Swagger` - za generiranje kode, dokumentacije in testov
- in `Spring Fox` - za Swagger integracijo s Spring aplikacijskim ogrodjem.

Kot je razvidno med zgoraj naštetimi knjižnicami, sem uporabil Spring Boot [22]. To je Spring knjižnica, ki omogoča izdelavo aplikacij, ki se zaženejo z vgrajenim Tomcat [2] strežnikom in se lahko konfigurira v celoti v Java kodi z uporabo konfiguracijskih nastavitev in anotacij. Če ne uporabljamo Spring Boot, lahko uporabimo za konfiguriranje tudi xml datoteke, vendar tega načina avtorji ne priporočajo več.

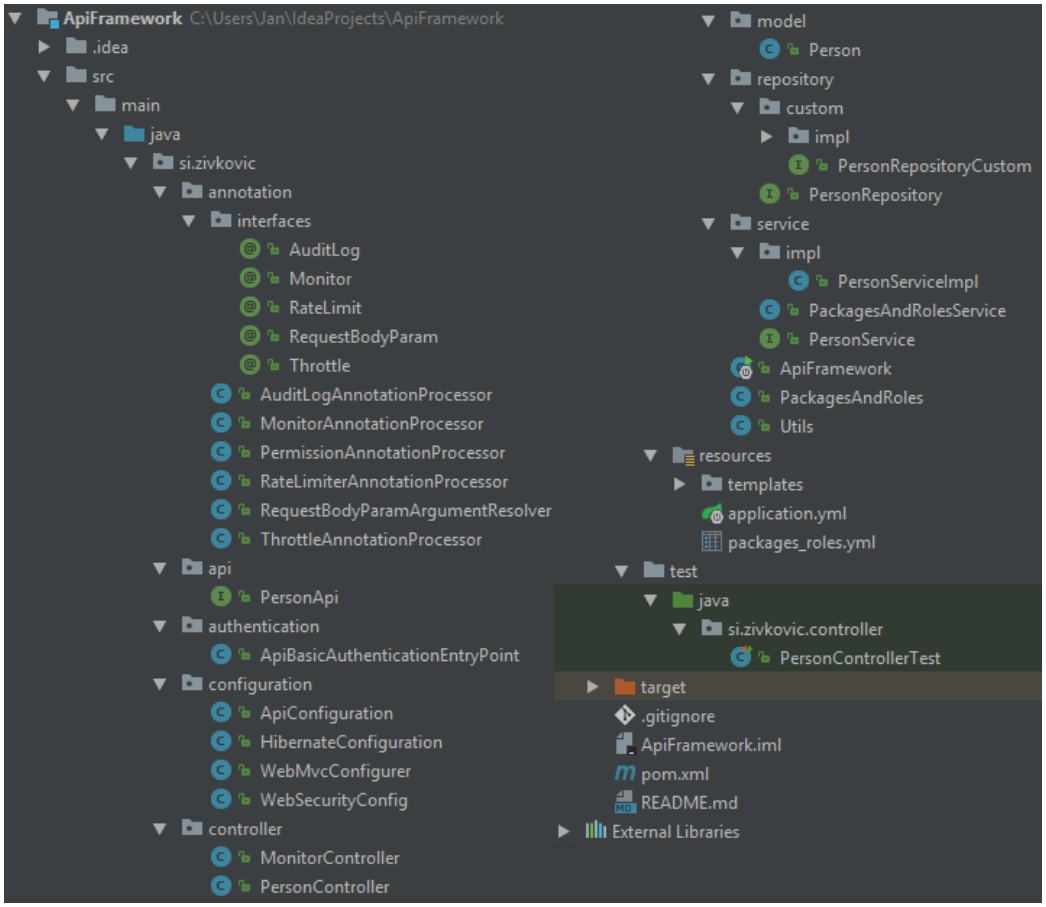
Hibernate je aplikacijsko ogrodje za objektno-relacijsko preslikovanje iz baze v objekte in s tem olajša delo, saj sam generira SQL stavke in vrača rezultate, mapirane v objekte, ki jih nato enostavno uporabimo v izvorni kodi.

MySQL-Connector je modul, ki projektu omogoči, da se Hibernate lahko poveže z MySQL bazo, v kateri hranimo podatke, ki jih nato aplikacija vrača preko REST servisa.

Apache Commons pa je modul, ki olajša delo s programskim jezikom Java tako, da vsebuje enostavne velikokrat uporabljene funkcije, kot so validacija podatkov, uporabne funkcije za manipuliranje besedila in objektov ter druga pogosta opravila. Apache Commons ni obvezen modul za izdelavo tega projekta, vendar je vključen zato, ker izredno poveča produktivnost programiranja v Java programskem jeziku.

4.2 Struktura aplikacije

Aplikacija je strukturirana tako, da je nadvse enostavna za uporabo in nadaljnji razvoj.



Slika 4.1: Struktura aplikacije

Kot je razvidno na sliki 4.1, je aplikacija razdeljena na več delov, kot so:

- `annotation` - vsebuje uporabljene anotacije in procesorje za anotacije,
- `api` - vsebuje definicijo in dokumentacijo,
- `authentication` - vsebuje razred za avtentikacijo,
- `configuration` - vsebuje vso konfiguracijo za Spring, Swagger in samo aplikacijo,
- `controller` - vsebuje kontrolerje, ki so vstopna točka za klice in vračajo podatke,
- `model` - vsebuje vse uporabljene modele,
- `repository` - vsebuje repozitorije za dostop do baze,
- `service` - vsebuje servise,
- `resources` - vsebuje vse potrebne vire, kot so konfiguracija za Spring, aplikacijo in Thymeleaf predloge,
- `test` - vsebuje teste za aplikacijo.

4.3 Postavitev MySQL baze

Za potrebe testiranja delovanja aplikacije sem uporabljal MySQL bazo, ki sem jo namestil s pomočjo XAMPP programa. Potem sem zagnal MySQL strežnik in kreiral bazo z imenom `test` z ukazom `CREATE DATABASE test;`, ki sem jo potem uporabljal za shranjevanje podatkov aplikacije. Potem ko je bila baza kreirana, sem lahko zagnal aplikacijo, kjer je Hibernate zaradi nastavitve `spring.jpa.hibernate.ddl-auto: update` ob zagonu samodejno posodobil ali kreiral potrebne tabele s pomočjo modelov, definiranih v aplikaciji.

4.4 Izdelava API aplikacije

Izdelavo API aplikacije sem začel s postavitvijo enostavnega Spring Boot projekta, ki je osnova celotne aplikacije. Po postavitvi projekta in testiranju, da projekt deluje, sem se lotil izdelave modelov. Modeli so objektna predstavitev zapisa v bazi, ki jih iz baze v objekt mapira Hibernate in se uporabljajo zaradi objektnega programiranja. Po uspešni izdelavi modela sem dodal Hibernate integracijo in začel z dodajanjem repozitorijev. Repozitoriji so enostavni vmesniki, ki omogočajo enostavne CRUD operacije nad modeli, kar pomeni, da vsebuje operacije, ki kreirajo, spreminjajo, berejo in brišejo zapise. Po preizkusu delovanja repozitorijev in same baze sem se lotil ustvarjanja servisov, ki so razredi, ki bodo izvajali vso logiko nad podatki. Za servisi sem ustvaril še API vmesnike in kontrolerje, kjer so vse vhodne točke za API. Nazadnje sem dodal še Spring security konfiguracijo, Swagger integracijo, anotacije, ki so bolje opisane v poglavju 5.4, in teste. Testi uporabljajo JUnit [14] knjižnico in se poganjajo z `@RunWith(SpringRunner.class)` ter anotacijo `@SpringBootTest`. Izvorna koda končanega primera API aplikacije je na voljo na naslednjem spletnem naslovu:
`https://drive.google.com/open?id=0B_vgeVfKfAtPWjlqWUZRUDJJcW8`

Poglavje 5

Izdelava generatorja za Swagger Codegen

Izvorna koda generatorja je na voljo na naslednjem spletnem naslovu: `https://drive.google.com/open?id=0B_vgeVfKfAtPSTdDT2RRalVTenc`

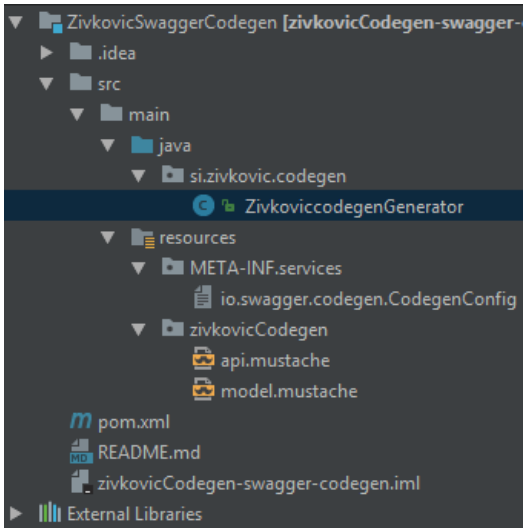
5.1 Generiranje osnovnega generatorja

Začel sem tako, da sem pregledal Swagger Codegen dokumentacijo, kjer imajo napisane napotke, kako naj bi se tega lotil. Swagger Codegen podpira generacijo začetnega projekta s strukturo, kot je v drugih generator modulih. Začetni projekt sem generiral s pomočjo Swagger Codegen projekta, ki sem ga kloniral z njihove GitHub [7] strani `https://github.com/swagger-api/swagger-codegen` in nato zgradil začetni projekt z ukazom, prikazanem v izseku kode 5.1.

Izsek kode 5.1: Ukaz za generiranje začetnega generator projekta

```
java -jar modules/swagger-codegen-cli/target/swagger
  ↪ -codegen-cli.jar meta -o "C:\Users\Jan\Desktop\
  ↪ Diploma\ZivkovicSwaggerCodegen" -n
  ↪ zivkovicCodegen -p si.zivkovic.codegen
```

Ukaz v izvorni kodi 5.1 je ustvaril projekt, ki je vseboval le en razred `ZivkoviccodegenGenerator`. To je glavni razred za generacijo projekta, ki sem ga kasneje preimenoval v `ZivkovicGenerator` zaradi predolgega imena. Vseboval je tudi `README.md`, kjer je bilo opisano, kako naj bi se lotil nadaljnje izdelave generatorja.



Slika 5.1: Struktura generiranega projekta za generator

`AbstractJavaCodegen` je osnovni razred v Swagger Codegen-u, ki se uporablja v osnovnih Java generatorjih in podpira precejšen del funkcionalnosti, ki bi jih sam uporabil. V konstruktorju sem nastavil osnovne spremenljivke, kot so paketi za kodo, podatki o artefaktu, uporabljenih knjižnicah in inicializiral celoten generator in implementiral preostale funkcije, ki se uporabljajo za generiranje.

5.2 Predloge za generator

Za generiranje projekta sem potreboval tudi predloge, iz katerih se bo koda generirala. Predloge so locirane v mapi `src/main/resources/zivkovicGenerator` ↪ . Vanjo sem dodal predloge za vse dele projekta, od konfiguracije do kode,

dokumentacije in testov. Nekaj teh datotek sem povzel iz že obstoječega Swagger Codegen generatorja `springCodegen`, saj sem tudi sam nameraval uporabiti Spring aplikacijsko ogrodje in se s tem izognil nepotrebnemu podvajanju kode. Ker sem v tem projektu vključil tudi druge knjižnice, kot je Hibernate, sem moral ustvariti in preurediti vse predloge, ki uporabljajo katerega od delov teh knjižnic. To je zajemalo predloge za modele, repozitorije, servise itd.
V prilogi A je prikazana datoteka, ki se uporablja za generiranje API vmesnika.

5.2.1 Statične predloge

Ker sem v projektu potreboval tudi statične razrede, ki vsebujejo konfiguracijo aplikacijskega ogrodja Spring in anotacij za prihodnje API ogrodje. Treba je bilo urediti še izvorno kodo in predloge za spremljanje delovanja aplikacije, avtentikacijo in en razred s pomožno metodo za sestavljanje unikatnih imen metod. Statične predloge sem dodal na način, prikazan v izseku kode 5.2.

Izsek kode 5.2: Prikaz dodajanja statičnih predlog

```
// Format dodajanja staticne predloge
//supportingFiles.add(new SupportingFile("
   ↪ imePredloge", "lokacijaKoncneDatoteke", "
   ↪ imeKoncneDatoteke"));

// Primer
supportingFiles.add(new SupportingFile("
   ↪ mvcConfigurer.mustache", configPath, "
   ↪ WebMvcConfigurer.java"));
```

S temi statičnimi predlogami sem v projekt dobil večino nujno potrebnih razredov za pravilno delovanje projekta in njegovo uporabo.

5.3 Paketi in pravice za dostop

Ker sem želel v aplikaciji imeti pakete in pravice za dostop do API, sem te definiral v yaml formatu v datoteki `packages\_roles.yml` po principu RBAC. RBAC je način dovoljevanja dostopa z uporabo vlog. S tem načinom sem lahko naredil več različnih vlog in uporabniku dodelil le tiste, ki jih potrebuje za dostop do potrebne funkcionalnosti. Tako sem enostavno, hitro in učinkovito preprečil dostop nepridipravom in s tem izboljšal varnost aplikacijskega ogrodja.

Paketi se uporabljajo za omejevanje dostopa glede do API tako, da omejujejo število vseh dostopov na uro in število hkratnih izvajajočih API klicev. S tem sem preprečil, da bi en sam uporabnik poslal toliko API klicev, da bi se API prenehal odzivati zaradi prevelikega števila klicev.

Za omejevanje dostopa do vsakega izmed klicev API se uporabljajo vloge. Primer omejevanja bi bil, da bi lahko določeni uporabniki samo brali podatke, drugi pa bi te podatke lahko tudi kreirali in spreminjali. Seveda se te vloge lahko nastavijo za vsakega uporabnika posebej. Tako lahko določimo pravice dostopa vsakemu uporabniku posebej. V spodnjem izseku kode 5.3 je prikazan primer datoteke za definiranje paketov in vlog.

Izsek kode 5.3: Primer datoteke packages_roles.yml

```
packages:
  FREE_PACKAGE:
    rateLimit: 1000
    throttleLimit: 1
  SMALL_PACKAGE:
    rateLimit: 2500
    throttleLimit: 5
  MEDIUM_PACKAGE:
    rateLimit: 5000
    throttleLimit: 10
  LARGE_PACKAGE:
```

```
    rateLimit: 10000
    throttleLimit: 10
  UNLIMITED_PACKAGE:
    rateLimit: -1
    throttleLimit: -1

roles:
  PERSON_CREATE:
    get:
      - /test/person
  PERSON_READ:
    get:
      - /test/listByName
      - /test/listByName2
    post:
      - /test/listByName2
  PERSON_UPDATE:
    get:
      - /test/updateById
  PERSON_DELETE:
    get:
      - /test/deleteById
```

Kot je razvidno iz izseka kode 5.3, je v datoteki definiranih 5 paketov. To so:

- `FREE_PACKAGE`,
- `SMALL_PACKAGE`,
- `MEDIUM_PACKAGE`,
- `LARGE_PACKAGE`,
- in `UNLIMITED_PACKAGE`.

Vsak paket ima svoje omejitve z izjemo `UNLIMITED_PACKAGE`, ki je neomejen. Iz zgornje datoteke so razvidne tudi vloge oz. pravice za dostop, ki se navezujejo na vsak posamezen API klic. Za klic na API `/test/listByName2`, je potrebna vloga `PERSON_READ`, s katero dobimo tudi klic na API `/test/` ↪ `listByName/`. Kot je razvidno v vlogi `PERSON_READ`, se omejuje dostop do API tudi glede na način klica metode, kjer so možni:

- `GET`,
- `POST`,
- `PUT`,
- in `DELETE`.

Na voljo so tudi drugi klici HTTP metod, vendar se v praksi ne uporabljajo oziroma se uporabljajo tako malo, da le redki vedo za njihov obstoj in jih zaradi tega nisem vključil.

5.3.1 Razred za preverjanje pravic

Ker sem za preverjanje pravic izdelal svojo konfiguracijsko datoteko, sem izdelal tudi razred, ki bi glede na to datoteko tudi preverjal pravice in omejeval dostop do API klicev. Ta razred je v izvorni kodi poimenovan `PermissionAnnotationProcessor.java` in preverja pravice za dostop samo na tistih metodah, ki so anotirane z anotacijo `@RequestMapping`. Ta anotacija se v Spring knjižnici uporablja za dostop do API klicev. Razred pravice preverja tako, da pridobi vse vloge trenutnega uporabnika in se nato sprehodi čez seznam vseh vlog in pripadajočih API klicev ter v primeru najdene metode dovoli dostop, drugače pa dostop zavrne s HTTP statusom 403 FORBIDDEN.

5.4 Anotacije in njihovi procesorji

V praksi je vedno potrebno tako omejevanje dostopa kot tudi zaviranje prepogostih dostopov do API klicev, beleženje in vodenje statistike API klicev. Zato sem dodal anotacije `@RateLimit`, `@Throttle` in `@Monitor`. Te anotacije za vodenje stanja uporabljajo objekte, ki se ponastavijo ob vsakem zagonu aplikacije. Za produkcijsko uporabo bi bilo treba te objekte zamenjati za bazo, kot je MySQL ali Redis [20], s čimer bi tudi podprli pravilno delovanje anotacij z večjim številom instanc te API aplikacije.

5.4.1 Anotacija za beleženje dostopov

V aplikacijah, ki so na voljo uporabnikom, je privzeto nastavljen najnižji nivo beleženja. To je `INFO` beleženje. Ker tudi v produkcijskih aplikacijah obstaja možnost napak, nam ta anotacija beleži začetek dostopa do API klica kot tudi konec dostopa do API klica na tem nivoju. To je uporabno, da se lažje ugotovi, kje in kdaj je prišlo do napake. Beleženje se vklopi za vsako metodo posebej z dodajanjem anotacije `@AuditLog` in izklopi z njeno odstranitvijo. Za izklop vsega beleženja je treba dodati nastavitev `auditLog` ↪ `.enabled=false` v datoteko `application.yml`.

5.4.2 Omejevanje dostopov do API na uro

V aplikacijah omejujemo dostope do API klicev zaradi preprečevanja prevelikega števila klicev enega uporabnika in posledično nedostopnost storitve drugim uporabnikom. S tem lahko zahtevnejšim strankam ponudimo storitev z višjimi omejitvami dostopov do API. Omejevanje dostopov do API na uro (angl. *rate limiting*) je aktivno samo na metodah, ki so anotirane z anotacijo `@RateLimit`, metode pa razvijalec anotira sam. Seveda se da omejevanje tudi izklopiti, kar naredimo z dodajanjem nastavitve `rateLimit.enabled=false` v datoteko `application.yml`. Anotacija tudi vedno doda tri vrednosti v glavo odgovora za potrebe obveščanja uporabnika o omejitvah. To so:

- `X-RateLimit-Limit` - število dovoljenih dostopov na uro,
- `X-RateLimit-Remaining` - preostalo število dostopov,
- in `X-RateLimit-Reset` - čas, ko se bo resetiralo število dostopov na uro.

V primeru, da uporabnik preseže število dovoljenih dostopov na uro, aplikacija vrne HTTP status 429 TOO MANY REQUESTS, s čimer lahko uporabnik s pregledom zgornjih vrednosti preveri, kdaj lahko spet uporablja dostop do API in tudi kakšne omejitve ima nastavljene, nato pa program prilagodi

tako, da teh omejitev ne doseže oziroma da ob dosegu omejitve preneha z API klici za čas, definiran v `X-RateLimit-Reset`.

5.4.3 Omejevanje hkratnih dostopov

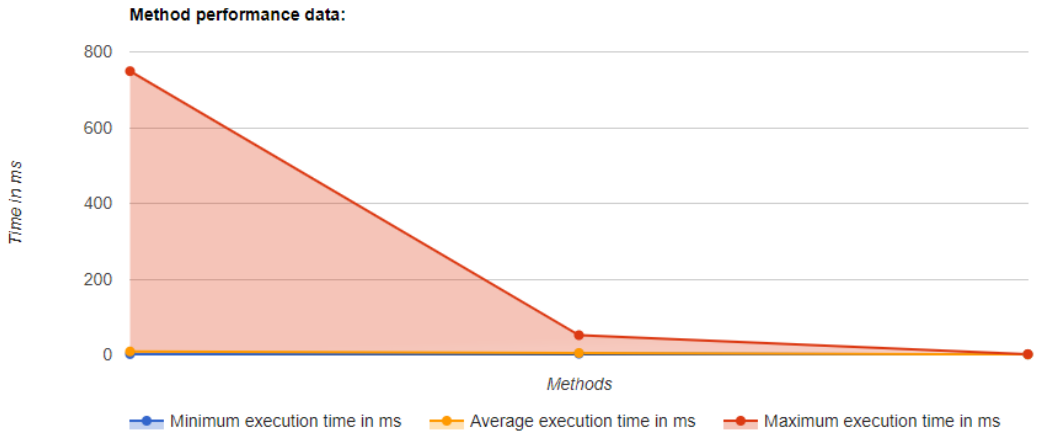
Omejevanje hkratnih dostopov (angl. *throttling*) zveni podobno kot omejevanje dostopov na uro, vendar ni tako. Omejevanje hkratnih dostopov omejuje le število hkratnih dostopov do istega API klica z istimi ali drugačnimi parametri. S tem se omeji prepogosto dostopanje do počasnih API klicev, ki potrebujejo veliko število sistemskih virov, in prepreči, da bi zmanjkalo sistemskih virov za druge uporabnike in druge klice. Tako kot pri omejevanju dostopov do API na uro se tudi za omejevanje hkratnih dostopov uporablja omejitev iz uporabnikovega paketa. Za vklop te funkcionalnosti na API klicu se doda anotacijo `@Throttle`. Funkcionalnost se za celotno aplikacijo lahko izklopi z dodajanjem nastavitve `throttle.enabled=false` v datoteki `application.yml`.

5.4.4 Spremljanje dostopov do API

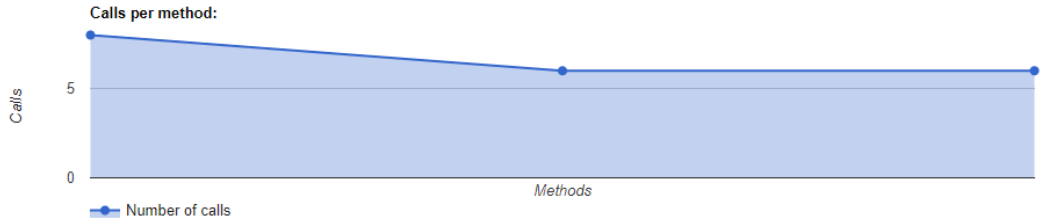
API vedno potrebuje tudi sistem za spremljanje in statistiko dostopov do API, da tako ugotovi, kateri API klic je najbolj popularen, kateri se odziva najpočasnejše, kateri se ne uporabljajo itd. Razvijalci s pregledom statistike lahko vidijo, katere API klice je treba izboljšati oz. popraviti, da bo API deloval čim boljše. Na trgu že obstajajo knjižnice, ki le-to podpirajo, kot so:

- `JAMon` [11]
- in `JavaMelody` [12],

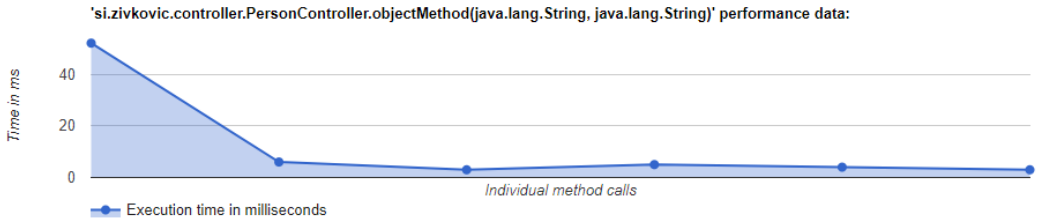
vendar so lahko zelo nepregledne in težavne za uporabo. Zato sem sam napisal anotacijo, ki prikaže le najbolj pomembne informacije. Rezultate vseh dostopov aplikacija prikaže vizualno z uporabo raznih grafov, ki prikazujejo čas izvajanja API klicev in njihovo število. Grafi so dostopni na url naslovu: `http://localhost:8080/v2/monitor`, primeri pa so prikazani na slikah 5.2, 5.3 in 5.4.



Slika 5.2: Prikaz časa izvajanja API klicev



Slika 5.3: Prikaz števila klicev API



Slika 5.4: Prikaz individualnih klicev API

Za prikaz grafov sem uporabil Google Charts [8] in sistem za generiranje predlog Thymeleaf [25].

5.4.5 RequestBodyParam anotacija

Dodal sem tudi anotacijo `@RequestBodyParam`, ki omogoča, da se vhodni xml ali json prebere in v spremenljivko vrine vrednost zahtevanega elementa. V anotaciji lahko podamo tri vrednosti, to so:

- `value` - naziv vrivanega elementa,
- `required` - `true`, če mora biti element nujno prisoten,
- in `defaultValue` - privzeta vrednost, če element ne obstaja.

Anotacija tudi preveri tip spremenljivke in element primerno preuredi v zahtevan tip.

Naslednja deklaracija `@RequestBodyParam(value = "/person/name")String name` vrine za vhodne podatke v xml (izsek kode 5.4) ali json (izsek kode 5.5) vrednost `Jan`.

Izsek kode 5.4: Primer vhodnih podatkov v xml formatu

```
<person>
        <name>Jan</name>
        <surname>Zivkovic</name>
        <age>22</age>
        <hobbies>
                <hobby>programming</hobby>
                <hobby>playing games</hobby>
                <hobby>watching movies</hobby>
        </hobbies>
</person>
```

Izsek kode 5.5: Primer vhodnih podatkov v json formatu

```
{
        "person": {
                "name": "Jan",
                "surname": "Zivkovic",
                "age": "22",
                "hobbies": {
                        "programming", "playing
                          ↪ games", "watching
                          ↪ movies"
                }
        }
}
```

5.5 Varnostna konfiguracija

Ker sem se odločil uporabiti Spring knjižnico, sem za varnost uporabil kar Spring security, ki je definiran v datoteki `WebSecurityConfig.java`. Ta datoteka vsebuje pravila za varnost, avtentikacijo in uporabnike. Njena vsebina je prikazana v prilogi C. Varnostna konfiguracija je omogočena z anotacijo `@EnableWebSecurity` in je nastavljena tako, da je za dostop do vseh API klicev potrebna avtentikacija z `basic auth`, ki zahteva uporabniško ime in geslo za dostop in je trenutno edina na voljo. Za avtentikacijo sta na voljo dva uporabnika. Uporabnik `ADMIN` ima vse pravice in neomejen paket `UNLIMITED_PACKAGE`, medtem ko ima uporabnik `USER` najmanjši paket `FREE_PACKAGE` in naslednje vloge za dostop:

- `PERSON_CREATE`,
- `PERSON_UPDATE`,
- in `PERSON_DELETE`.

S temi vlogami lahko v datoteki, prikazani v izseku kode 5.3, preverimo, do katerih API klicev ima uporabnik dostop.

5.6 Problemi

Med izdelavo sem naletel tudi na nekaj problemov, ki sem jih tudi uspešno rešil. Spodaj so prikazani različni problemi, s katerimi sem se soočil med izdelovanjem aplikacije.

5.6.1 Generiranje repozitorijev in servisov

Za generiranje repozitorijev in servisov sem imel skoraj vse podatke že vsebovane v kodi, ki je bila zadolžena za generiranje modelov. Zato sem se odločil, da bom repozitorije in servise poskusil generirati z uporabo sistema za generiranje modelov. Kmalu sem pristal na trdnih realnih tleh, ko nisem našel načina, kako bi generiral datoteke za repozitorije in servise v svoji mapi in ne v mapi, ki je namenjena modelom. Za reševanje problema sem porabil veliko časa in raziskovanja, kako je obstoječa koda spisana in ali obstaja kje kakšna luknja, s pomočjo katere bi lahko datoteke generiral drugje kot v privzeti mapi. Po mučnem in dolgotrajnem iskanju sem jo našel in je prikazana v izseku kode 5.6, kjer vidimo, kako lahko z manipuliranjem imena datoteke menjavamo mapo. V izseku kode 5.7 je viden način, kako sem to luknjo uspešno izkoristil. Ta luknja deluje zato, ker v konstruktorju razreda `File` Java dopušča uporabo relativne ali absolutne poti do datoteke.

Izsek kode 5.6: Izkoriščen del kode v datoteki AbstractGenerator.java

```
public File writeToFile(String filename, String
   ↪ contents) throws IOException {
  LOGGER.info("writing␣file␣" + filename);
  File output = new File(filename);

  if (output.getParent() != null && !new File(output
     ↪ .getParent()).exists()) {
    File parent = new File(output.getParent());
    parent.mkdirs();
  }
  Writer out = new BufferedWriter(new
     ↪ OutputStreamWriter(
```

```
  new FileOutputStream(output), "UTF-8"));

  out.write(contents);
  out.close();
  return output;
}
```

Izsek kode 5.7: Manipuliranje imena datoteke za menjavanje mape

```
@Override
public String toModelFilename(String name) {
  final String modelFilename = super.toModelFilename
    ↪ (name);
  Integer called = modelFilenameCalled.get(name);
  called = called == null? 1: called + 1;
  modelFilenameCalled.put(name, called);
  switch (called){
    case 1: // Never printed ???
      return modelFilename;
    case 2: // Repository.java
      return "../repository/" + modelFilename;
    case 3:// Model.java
      return modelFilename;
    case 4:// RepositoryImpl.java
      return "../repository/custom/impl/" +
        ↪ modelFilename;
    case 5:// CustomRepository.java
      return "../repository/custom/" + modelFilename
        ↪ ;
    case 6:// Service.java
      return "../service/" + modelFilename;
    case 7:// ServiceImpl.java
      return "../service/impl/" + modelFilename;
    default:
      return modelFilename;
  }
}
```

5.6.2 Generiranje datoteke packages_roles.yml

Ker sem želel, da bi se tudi konfiguracijska datoteka za pakete in vloge generirala poleg preostale izvorne kode, je nastala težava, ko sem moral uporabiti sicer že prebrano konfiguracijsko datoteko, saj le-ta v generatorju privzeto ni dostopna. Po dolgem raziskovanju delovanja generiranja izvorne kode sem ugotovil, da do vsebine konfiguracijske datoteke dostopam le z njeno celotno potjo in imenom. S tem sem lahko dostopil do njene vsebine in generiral še manjkajočo datoteko s paketi in vlogami.

Za branje datoteke sem uporabil izsek kode 5.8 in prebrano konfiguracijo z uporabo kode v prilogi D transformiral v primerne objekte ter jih tako uporabil v predlogi za generacijo datoteke.

Izsek kode 5.8: Branje Swagger specifikacije API

```
final Swagger swagger = new SwaggerParser().read(
    ↪ inputSpec, null, true);
```

Poglavje 6

Uporaba generatorja

V tem poglavju je na kratko opisano, kako se uporabnik loti generiranja API projekta s konfiguracijsko datoteko in kako se loti kasnejšega dopolnjevanja izvorne kode.

6.1 Konfiguracijska datoteka

Konfiguracijska datoteka je glavna datoteka, na podlagi katere generator generira celotno aplikacijo. V tej konfiguraciji so definirane informacije API-ja, API klici, vhodni in izhodni parametri API klicev, dokumentacija, vloge, paketi in še veliko drugega. Primer konfiguracijske datoteke je prikazan v prilogi B.

V konfiguracijsko datoteko sem dodal tudi nove lastnosti, ki se morajo po Swagger specifikaciji začeti z `x-`, te pa so:

- `x-permission-role` - dodajanje vloge za dostop do API klica,
- `x-hibernate-tableName` - ime generirane tabele za model,
- in `x-hibernate-primaryKey` - označba za primarni ključ v bazi.

Te lastnosti so obvezne za pravilno generiranje izvorne kode.

6.2 Generiranje API s konfiguracijsko datoteko

Generiranje izvedemo z ukazom v izseku kode 6.1, kjer se v ukazu zamenja `{apiKonfiguracijskaDatoteka.yml}` s potjo do konfiguracijske datoteke in `{izhodnaMapa}` z mapo, kjer naj generator generira datoteke projekta. Ko je projekt generiran, ga lahko vključimo v IDE in poženemo proceduro za grajenje projekta, ali ga zgradimo z uporabo Maven ukaza `mvn clean install`.

Izsek kode 6.1: Ukaz za generiranje API

```
java -cp target/zivkovicCodegen-swagger-codegen
    -1.0.0.jar;lib/swagger-codegen-cli.jar io.
    swagger.codegen.SwaggerCodegen generate -l
    zivkovicCodegen -i {apiKonfiguracijskaDatoteka.
    yml} -o {izhodnaMapa}
```

Ko je projekt generiran in dodan v poljuben IDE oziroma urejevalnik izvorne kode, lahko začnemo s spreminjanjem izvorne kode in dodajanjem implementacije vseh API klicev. Kontrolerji so vstopna točka vsakega API klica in podatke le posredujejo naprej servisom z implementacijo API klicev. Za dostop do baze, na kateri operirajo API klici, se uporabljajo repozitoriji. Zanje lahko v vmesniku `{ime}Repository.java` definiramo nove metode in SQL poizvedbe, kot je prikazano v izseku kode 6.2. Tu vidimo, kako se v `@Query` napiše poizvedbo in v `@Param` ime spremenljivke v uporabljenem modelu.

Za dodajanje zahtevnejših poizvedb v bazi je dodan tudi poseben razred `{ime}RepositoryCustomImpl.java`, kjer se lahko v Java kodi definira poizvedbe z večjo fleksibilnostjo kot v samem vmesniku z anotacijo. Za dodajanje ali spreminjanje paketov in vlog se uporablja datoteka `packages_roles.yml`, za spreminjanje dokumentacije pa vmesniki `{ime}Api.java`, kjer se `{ime}` zamenja z imenom vsakega modela.

Izsek kode 6.2: Primer dodajanja SQL poizvedbe z anotacijo

```
@Query("SELECT p FROM Person p WHERE LOWER(p.
    ↪ lastName) = LOWER(:lastName)")
public List<Person> find(@Param("lastName") String
    ↪ lastName);
```

6.3 Predkonfiguracija in zagon aplikacije

Aplikaciji je pred zagonom treba spremeniti določene nastavitve v datoteki `application.yml`. Nastavitve, ki jih je nujno treba spremeniti, so ime baze, uporabniško ime in geslo za dostop do baze. Druge nastavitve lahko pustimo nedotaknjene.

6.4 Pregled dokumentacije

Po zagonu aplikacije lahko tudi pregledamo dokumentacijo API na spletnem naslovu `http://localhost:8080/v2`, obstaja pa tudi dokumentacija v tekstovnem formatu v mapi `{mapaZgeneriranegaProjekta}/docs`.

Poglavje 7

Zaključek

V diplomskem delu sem opisal izdelavo aplikacije, s katero lahko iz enostavne konfiguracijske datoteke generiramo aplikacijo z delujočim API vmesnikom, pripadajočo izvorno kodo in konfiguracijo, ki je potrebna za delovanje. Pri tem sem uporabil knjižnice Spring, Hibernate in Swagger ter z njimi pospešil izdelavo aplikacije. Aplikacija sama pa pospeši prototipiranje in razvoj novih spletnih aplikacij. Obenem podpira tudi spremljanje delovanja, preprečevanje dostopa do API klicev glede na vloge in omejevanje dostopa glede na paket uporabnika.

Projekt bi lahko dodatno izboljšal tako, da bi dodal še bazo Redis, jo uporabil v anotacijah `@RateLimit`, `@Throttle`, `@Monitor` in s tem omogočil, da bi delovale pravilno tudi v primeru sočasnega izvajanja več instanc aplikacije, saj bi bile vse omejitve in podatki za spremljanje delovanja shranjeni v bazi in ne v objektih v vsaki instanci aplikacije posebej.

Menim, da je cilj diplomskega dela dosežen, saj je aplikacija uporabna za generiranje projektov v Java izvorni kodi z uporabo že obstoječih knjižnic in je s tem pohitrila prototipiranje in izdelovanje le-teh.

Literatura

[1] Apache log4j 2. Dosegljivo: https://logging.apache.org/log4j/2.x/. [Dostopano: 7. 5. 2017].

[2] Apache tomcat. Dosegljivo: http://tomcat.apache.org/. [Dostopano: 7. 5. 2017].

[3] Api studio. Dosegljivo: http://apistudio.io/. [Dostopano: 6. 5. 2017].

[4] Apiary. Dosegljivo: https://apiary.io/. [Dostopano: 6. 5. 2017].

[5] Git. Dosegljivo: https://git-scm.com/. [Dostopano: 7. 5. 2017].

[6] Git bash. Dosegljivo: https://git-for-windows.github.io/. [Dostopano: 7. 5. 2017].

[7] Github. Dosegljivo: https://github.com/. [Dostopano: 24. 6. 2017].

[8] Google charts. Dosegljivo: https://developers.google.com/chart/. [Dostopano: 24. 6. 2017].

[9] Hibernate. Dosegljivo: http://hibernate.org/. [Dostopano: 7. 5. 2017].

[10] Intellij idea. Dosegljivo: https://www.jetbrains.com/idea/. [Dostopano: 7. 5. 2017].

[11] Jamon. Dosegljivo: http://jamonapi.sourceforge.net/spring_aop_monitoring.html. [Dostopano: 8. 7. 2017].

[12] Javamelody. Dosegljivo: https://github.com/javamelody/javamelody/wiki/UserGuide. [Dostopano: 8. 7. 2017].

[13] Json. Dosegljivo: http://www.json.org/. [Dostopano: 13. 5. 2017].

[14] Junit. Dosegljivo: http://junit.org/junit4/. [Dostopano: 7. 5. 2017].

[15] Mashape. Dosegljivo: https://www.mashape.com/. [Dostopano: 6. 5. 2017].

[16] Maven. Dosegljivo: https://maven.apache.org/. [Dostopano: 7. 5. 2017].

[17] Mustache. Dosegljivo: https://mustache.github.io/. [Dostopano: 9. 7. 2017].

[18] Notepad++. Dosegljivo: https://notepad-plus-plus.org/. [Dostopano: 7. 5. 2017].

[19] Postman. Dosegljivo: https://www.getpostman.com/. [Dostopano: 13. 5. 2017].

[20] Redis. Dosegljivo: https://redis.io/. [Dostopano: 8. 7. 2017].

[21] Spring. Dosegljivo: https://spring.io/. [Dostopano: 7. 5. 2017].

[22] Spring boot. Dosegljivo: https://projects.spring.io/spring-boot/. [Dostopano: 7. 5. 2017].

[23] Springfox. Dosegljivo: http://springfox.github.io/springfox/. [Dostopano: 7. 5. 2017].

[24] Swagger. Dosegljivo: https://swagger.io/. [Dostopano: 6. 5. 2017].

[25] Thymeleaf. Dosegljivo: http://www.thymeleaf.org/. [Dostopano: 24. 6. 2017].

[26] Xampp. Dosegljivo: https://www.apachefriends.org/index.html. [Dostopano: 27. 5. 2017].

[27] Yaml. Dosegljivo: http://yaml.org/. [Dostopano: 13. 5. 2017].

Priloge

Priloga A

Datoteka api.mustache, uporabljena za generiranje API vmesnika

```
package {{package}};

{{#imports}}import {{import}};
{{/imports}}

import io.swagger.annotations.*;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestPart;
import org.springframework.web.multipart.MultipartFile;
import java.io.IOException;

import java.util.List;
{{#async}}
import java.util.concurrent.{{^jdk8}}Callable{{/jdk8}}{{#jdk8}}
    ↪ CompletableFuture{{/jdk8}};
{{/async}}
{{#useBeanValidation}}
import javax.validation.constraints.*;
import javax.validation.Valid;
{{/useBeanValidation}}
```

```
{{>generatedAnnotation}}
@Api(value = "{{{baseName}}}", description = "the {{{baseName}}}
    ↪ API")
{{#operations}}
public interface {{classname}} {
{{#operation}}

    @ApiOperation(value = "{{{summary}}}", notes = "{{{notes}}}"
        ↪ , response = {{{returnType}}}.class{{#returnContainer
        ↪ }}, responseContainer = "{{{returnContainer}}}"{{/
        ↪ returnContainer}}{{#hasAuthMethods}}, authorizations =
        ↪  {
        {{#authMethods}}@Authorization(value = "{{name}}"{{#
            ↪ isOAuth}}, scopes = {
            {{#scopes}}@AuthorizationScope(scope = "{{scope}}",
                ↪ description = "{{description}}"){{#hasMore}},
            {{/hasMore}}{{/scopes}}
            }{{/isOAuth}}){{#hasMore}},
        {{/hasMore}}{{/authMethods}}
    }{{/hasAuthMethods}}, tags={ {{#vendorExtensions.x-tags}}"{{
        ↪ tag}}",{{/vendorExtensions.x-tags}}
    })
    @ApiResponses(value = { {{#responses}}
        @ApiResponse(code = {{{code}}}, message = "{{{message}}}
            ↪ ", response = {{{returnType}}}.class){{#hasMore
            ↪ }},{{/hasMore}}{{/responses}}
    })
    {{#implicitHeaders}}
    @ApiImplicitParams({
    {{#headerParams}}{{>implicitHeader}}{{/headerParams}}
    })
    {{/implicitHeaders}}
    @RequestMapping(value = "{{{path}}}",{{#singleContentTypes}}
        produces = "{{{vendorExtensions.x-accepts}}}",
        consumes = "{{{vendorExtensions.x-contentType}}}",{{/
            ↪ singleContentTypes}}{{^singleContentTypes}}{{#
            ↪ hasProduces}}
        produces = { {{#produces}}"{{{mediaType}}}"{{#hasMore}},
            ↪  {{/hasMore}}{{/produces}} }, {{/hasProduces}}{{#
            ↪ hasConsumes}}
        consumes = { {{#consumes}}"{{{mediaType}}}"{{#hasMore}},
            ↪  {{/hasMore}}{{/consumes}} }, {{/hasConsumes}}{{/
            ↪ singleContentTypes}}
        method = RequestMethod.{{httpMethod}})
    default {{#responseWrapper}}{{.}}<{{/responseWrapper}}
        ↪ ResponseEntity<{{>returnTypes}}>{{#responseWrapper
        ↪ }}>{{/responseWrapper}} {{operationId}}({{#allParams
        ↪ }}{{>queryParams}}{{>pathParams}}{{>headerParams}}{{>
        ↪ bodyParams}}{{>formParams}}{{^-last}},{{/-last}}{{/
        ↪ allParams}})){{#examples}}{{#-first}} throws
        ↪ IOException{{/-first}}{{/examples}} {
        // do some magic!
```

```
        return {{#async}}CompletableFuture.completedFuture({{/
            ↪ async}}new ResponseEntity<{{>returnTypes}}>(
            ↪ HttpStatus.OK){{#async}}){{/async}};
    }

{{/operation}}
}
{{/operations}}
```

Priloga B

Konfiguracijska datoteka

```
swagger: '2.0'
info:
  description: Example
  version: 1.0.0
  title: Swagger Example
  termsOfService: 'http://swagger.io/terms/'
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
host: localhost
basePath: /v2
tags:
  - name: user
    description: Operations about user
    externalDocs:
      description: Find out more about swagger
      url: 'http://swagger.io'
schemes:
  - http
paths:
  /user:
    post:
      x-permission-role: PERSON_CREATE
      tags:
        - user
      summary: Create user
      description: This can only be done by the logged in user.
      operationId: createUser
      parameters:
        - in: body
          name: body
          description: Created user object
```

```
        required: true
        schema:
          $ref: '#/definitions/User'
    responses:
      default:
        description: successful operation
/user/createWithArray:
  post:
    x-permission-role: PERSON_CREATE
    tags:
      - user
    summary: Creates list of users with given input array
    description: ''
    operationId: createUsersWithArrayInput
    parameters:
      - in: body
        name: body
        description: List of user object
        required: true
        schema:
          type: array
          items:
            $ref: '#/definitions/User'
    responses:
      default:
        description: successful operation
/user/createWithList:
  post:
    x-permission-role: PERSON_CREATE
    tags:
      - user
    summary: Creates list of users with given input array
    description: ''
    operationId: createUsersWithListInput
    parameters:
      - in: body
        name: body
        description: List of user object
        required: true
        schema:
          type: array
          items:
            $ref: '#/definitions/User'
    responses:
      default:
        description: successful operation
/user/login:
  get:
    x-permission-role: PERSON_READ
    tags:
      - user
    summary: Logs user into the system
```

```
    description: ''
    operationId: loginUser
    parameters:
      - name: username
        in: query
        description: The user name for login
        required: true
        type: string
      - name: password
        in: query
        description: The password for login in clear text
        required: true
        type: string
    responses:
      '200':
        description: successful operation
        schema:
          type: string
      '400':
        description: Invalid username/password supplied
/user/logout:
  get:
    x-permission-role: PERSON_READ
    tags:
      - user
    summary: Logs out current logged in user session
    description: ''
    operationId: logoutUser
    parameters: []
    responses:
      default:
        description: successful operation
'/user/{username}':
  get:
    x-permission-role: PERSON_READ
    tags:
      - user
    summary: Get user by user name
    description: ''
    operationId: getUserByName
    parameters:
      - name: username
        in: path
        description: 'The name that needs to be fetched. Use
          ↪ user1 for testing. '
        required: true
        type: string
    responses:
      '200':
        description: successful operation
        schema:
          $ref: '#/definitions/User'
```

```
        '400':
          description: Invalid username supplied
        '404':
          description: User not found
    put:
      x-permission-role: PERSON_UPDATE
      tags:
        - user
      summary: Updated user
      description: This can only be done by the logged in user.
      operationId: updateUser
      parameters:
        - name: username
          in: path
          description: name that need to be updated
          required: true
          type: string
        - in: body
          name: body
          description: Updated user object
          required: true
          schema:
            $ref: '#/definitions/User'
      responses:
        '400':
          description: Invalid user supplied
        '404':
          description: User not found
    delete:
      x-permission-role: PERSON_DELETE
      tags:
        - user
      summary: Delete user
      description: This can only be done by the logged in user.
      operationId: deleteUser
      parameters:
        - name: username
          in: path
          description: The name that needs to be deleted
          required: true
          type: string
      responses:
        '400':
          description: Invalid username supplied
        '404':
          description: User not found
definitions:
  User:
    type: object
    x-hibernate-tableName: 'user'
    properties:
      id:
```

```
        x-hibernate-primaryKey: true
        type: integer
        format: int64
      username:
        type: string
      firstName:
        type: string
      lastName:
        type: string
      email:
        type: string
      password:
        type: string
      phone:
        type: string
      userStatus:
        type: integer
        format: int32
        description: User Status
    xml:
      name: User
externalDocs:
  description: Find out more about Swagger
  url: 'http://swagger.io'
x-packages:
  FREE_PACKAGE:
    rateLimit: 1000
    throttleLimit: 1
  SMALL_PACKAGE:
    rateLimit: 2500
    throttleLimit: 5
  MEDIUM_PACKAGE:
    rateLimit: 5000
    throttleLimit: 10
  LARGE_PACKAGE:
    rateLimit: 10000
    throttleLimit: 10
  UNLIMITED_PACKAGE:
    rateLimit: -1
    throttleLimit: -1
```

Priloga C

WebSecurityConfig.java

```
package si.zivkovic.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.
    ↪ authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.
    ↪ builders.HttpSecurity;
import org.springframework.security.config.annotation.web.
    ↪ configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.
    ↪ configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.
    ↪ SessionCreationPolicy;
import si.zivkovic.authentication.
    ↪ ApiBasicAuthenticationEntryPoint;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
    ↪ WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .authorizeRequests()
        .anyRequest()
        .authenticated()
        .and()
      .formLogin()
        .disable()
      .logout()
        .permitAll()
        .and()
```

```
        .csrf()
          .disable()
        .httpBasic()
          .realmName(ApiBasicAuthenticationEntryPoint.REALM_NAME)
          .authenticationEntryPoint(
              ↪ apiBasicAuthenticationEntryPoint())
          .and()
        .sessionManagement()
          .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        ↪ throws Exception {
      auth.inMemoryAuthentication()
        .withUser("admin").password("admin").roles("ADMIN", "
            ↪ UNLIMITED_PACKAGE")
        .and()
        .withUser("user").password("user").roles("USER", "
            ↪ PERSON_CREATE", "PERSON_UPDATE", "PERSON_DELETE", "
            ↪ FREE_PACKAGE");
    }

    @Bean
    public ApiBasicAuthenticationEntryPoint
        ↪ apiBasicAuthenticationEntryPoint(){
      return new ApiBasicAuthenticationEntryPoint();
    }

}
```

Priloga D

Transformiranje Swagger specifikacije v objekte in generiranje datoteke packages_roles.yml

```
private void addPackagesAndRoles(final Swagger swagger, final
    ↪ String resourcesPath){
  // Packages
  final Map<String, Object> vendorExtensions = swagger.
      ↪ getVendorExtensions();
  final Map<String, Map<String, Integer>> packages = (Map<String
      ↪ , Map<String, Integer>>) vendorExtensions.get("x-
      ↪ packages");
  additionalProperties.put("packages", packages.entrySet().
      ↪ stream()
       .collect(Collectors.toMap(
           Map.Entry::getKey, entry -> entry.getValue().entrySet
               ↪ ()))
       ).entrySet()
  );

  // Roles
  final Map<String, Map<String, List<String>>> roles = new
      ↪ TreeMap<>();
  for(Map.Entry<String, Path> entryPath: swagger.getPaths().
      ↪ entrySet()){
    final Path path = entryPath.getValue();
    for(Map.Entry<HttpMethod, Operation> entryMethod: path.
        ↪ getOperationMap().entrySet()){
```

```
        final String method = entryMethod.getKey().name().
            ↪ toLowerCase();
        final Operation operation = entryMethod.getValue();
        final String pathPermissionRole = (String) operation.
            ↪ getVendorExtensions().get("x-permission-role");
        if(!roles.containsKey(pathPermissionRole)){
          roles.put(pathPermissionRole, new TreeMap<>());
        }
        if(!roles.get(pathPermissionRole).containsKey(method)){
          roles.get(pathPermissionRole).put(method, new ArrayList
              ↪ <>());
        }
        roles.get(pathPermissionRole).get(method).add(entryPath.
            ↪ getKey());
      }
    }

    additionalProperties.put("roles", roles.entrySet().stream()
        .collect(Collectors.toMap(
            Map.Entry::getKey, entry -> entry.getValue().entrySet
                ↪ ()))
        ).entrySet()
    );
    supportingFiles.add(new SupportingFile("packages_roles.
        ↪ mustache", resourcesPath, "packages_roles.yml"));
}
```